Inversion of a recursive tree traversal

Jan L.A. van de Snepscheut
Computer Science
California Institute of Technology
Pasadena, CA 91125

10 May 1991

Keywords program inversion, recursion, infix traversal, prefix traversal.

Abstract A recursive algorithm for generating the prefix and infix traversals of a binary tree is inverted to obtain an algorithm for constructing the tree from its traversals.

1. Introduction

Constructing a labeled binary tree from its infix and prefix traversal is a topic that shows up every now and then in places ranging from the writings of Knuth (cf. [7]) to the ACM Programming Competition. Solutions exist that are very different at first sight, although the differences becomes less pronounced at second sight. One class of solutions is presented in two steps: a simple iterative algorithm for producing the infix and prefix traversal of a tree is given and then inverted to obtain a solution to the inverse problem. A direct proof of the latter algorithm is nontrivial whereas the rules of inversion guarantee its correctness since the forward algorithm is correct. The other class of solutions consists of recursive procedures that are "seen" in an instant. This note does not provide yet another recursive solution. (Despite the fact that inefficient solutions have been published (cf. [1,2]), an efficient recursive solution is almost trivial, and has been produced independently by various authors, but was never published.) Instead, we show how the recursive algorithm can be cast in the framework of program inversion.

2. Generating the traversals of a tree

A labeled binary tree is either

$\perp$ if it is the empty tree, or

$\langle t.l, t.d, t.r \rangle$ if it is a nonempty tree t with left and right subtrees $t.l$ and $t.r$ and label $t.d$.

In order to be able to construct the tree from its traversals it is required that the labels in the tree are distinct. Infix traversal $in(t)$ and prefix traversal $pre(t)$ of tree t are readily stored in sequence variables x and y through execution of the following algorithm.

$x, y := \epsilon, \epsilon;\ gen(t)$

where

procedure $gen(t : tree)$:
  if $t = \perp \rightarrow skip$
  $[\!]$ $t \neq \perp \rightarrow y := y +\!\!+ t.d;\ gen(t.l);\ x := x +\!\!+ t.d;\ gen(t.r)$
  fi

We write ϵ for the empty sequence and $+$ for catenating sequences. We do not distinguish between a sequence of one element and the element itself. The specification of the procedure is

$$\{\ x = X \ \wedge \ y = Y\ \}$$
$$gen(t)$$
$$\{\ x = X +\!\!+ in(t) \ \wedge \ y = Y +\!\!+ pre(t)\ \} \quad .$$

In the sequel we write the pre- and postcondition of a procedure before and after the procedure body. In order to obtain the program that we are after, we change the above program to produce the sequences x and y from right to left instead of from left to right.

procedure $gen(t : tree)$:
$\{\ x = X \ \wedge \ y = Y\ \}$
  if $t = \perp \rightarrow skip$
  $[\!]$ $t \neq \perp \rightarrow gen(t.r);\ x := t.d +\!\!+ x;\ gen(t.l);\ y := t.d +\!\!+ y$
  fi
$\{\ x = in(t) +\!\!+ X \ \wedge \ y = pre(t) +\!\!+ Y\ \}$

We verify the correctness of this procedure against its specification. Since the program is recursive, the proof is by mathematical induction: in the proof it can be assumed that the recursive calls satisfy their specificationbecause their arguments $t.r$ and $t.l$ are proper subtrees of tree t. There are two cases. If $t = \perp$ then

$$\begin{array}{ll} & wp(skip, x = in(t) +\!\!+ X \ \wedge \ y = pre(t) +\!\!+ Y) \\ = & \quad \{\ in(\perp) = \epsilon, pre(\perp) = \epsilon\ \} \\ & wp(skip, x = X \ \wedge \ y = Y) \\ = & \quad \{\ \text{definition of } skip\ \} \\ & x = X \ \wedge \ y = Y \end{array}$$

If $t \neq \perp$ we use the abbreviation s_k for the first k statements in

$$gen(t.r);\ x := t.d +\!\!+ x;\ gen(t.l);\ y := t.d +\!\!+ y$$

and calculate

$$\begin{array}{ll} & wp(s_4, x = in(t) +\!\!+ X \ \wedge \ y = pre(t) +\!\!+ Y) \\ = & \quad \{\ in(t) = in(t.l) +\!\!+ t.d +\!\!+ in(t.r), pre(t) = t.d +\!\!+ pre(t.l) +\!\!+ pre(t.r)\ \} \\ & wp(s_4, x = in(t.l) +\!\!+ t.d +\!\!+ in(t.r) +\!\!+ X \ \wedge \ y = t.d +\!\!+ pre(t.l) +\!\!+ pre(t.r) +\!\!+ Y) \\ \Leftarrow & \quad \{\ \text{rule of assignment}\ \} \\ & wp(s_3, x = in(t.l) +\!\!+ t.d +\!\!+ in(t.r) +\!\!+ X \ \wedge \ y = pre(t.l) +\!\!+ pre(t.r) +\!\!+ Y) \\ \Leftarrow & \quad \{\ \text{specification of } gen(t.l)\ \} \\ & wp(s_2, x = t.d +\!\!+ in(t.r) +\!\!+ X \ \wedge \ y = pre(t.r) +\!\!+ Y) \\ \Leftarrow & \quad \{\ \text{rule of assignment}\ \} \\ & wp(s_1, x = in(t.r) +\!\!+ X \ \wedge \ y = pre(t.r) +\!\!+ Y) \\ \Leftarrow & \quad \{\ \text{specification of } gen(t.r)\ \} \\ & x = X \ \wedge \ y = Y \quad . \end{array}$$

3. Generating the tree from its traversals

Inversion of the program in the previous section requires that we come up with mutually exclusive postconditions for the two alternatives in the if-statement (cf. [4,5]). It seems, however, that no simple conditions exist. Operationally speaking, the problem is that all work is done in the second alternative; no variables change in the first alternative so that it is hard to "detect" those calls that select the first alternative. Therefore, we propose to shift some of the work from the second to the first alternative. Since the statement $y := t.d +\!\!+ y$ is the last state change of the second alternative it is probably indicative of which alternative was chosen and we don't want to lose that information. Therefore, we move the assignment to x to the first alternative. The problem in doing so, however, is that the label to be prefixed to sequence x is not available in this situation, so we add it as a parameter to the procedure.

$$
\begin{array}{l}
\textbf{procedure}\ gen(d : label;\ t : tree) : \\
\{\ x = X \quad \wedge \quad y = Y\ \} \\
\quad \textbf{if}\ t = \bot \rightarrow x := d +\!\!+ x \\
\quad [\!]\ \ t \neq \bot \rightarrow gen(d, t.r);\ gen(t.d, t.l);\ y := t.d +\!\!+ y \\
\quad \textbf{fi} \\
\{\ x = in(t) +\!\!+ d +\!\!+ X \quad \wedge \quad y = pre(t) +\!\!+ Y\ \}
\end{array}
$$

The program consists of

$$x, y := \epsilon, \epsilon;\ gen(\emptyset, t)$$

in which $\emptyset$ is a label that does not occur in tree t and that is appended to $in(t)$ in sequence x. A postcondition of the first alternative is $hd(x) = d$, where $hd(x)$ is the first element of sequence x. We write $tl(x)$ for the remainder of sequence x. A postcondition of the second alternative is $hd(x) = t.d$. The two postconditions are disjoint if $d \neq t.d$ and this is in turn implied by the condition that all labels in the tree are distinct. Inversion of the above program yields procedure neg which stores in t the tree whose traversals are given in x and y. It requires that x be extended with a label that does not occur anywhere else in the two traversals.

$$
\begin{array}{l}
\textbf{procedure}\ neg(d : label;\ \textbf{var}\ t : tree) : \\
\{\ x = in(t) +\!\!+ d +\!\!+ X \quad \wedge \quad y = pre(t) +\!\!+ Y\ \} \\
\quad \textbf{if}\ hd(x) = d \rightarrow t, x := \bot, tl(x) \\
\quad [\!]\ \ hd(x) \neq d \rightarrow t.d, y := hd(y), tl(y);\ neg(t.d, t.l);\ neg(d, t.r) \\
\quad \textbf{fi} \\
\{\ x = X \quad \wedge \quad y = Y\ \}
\end{array}
$$

The program is

$$
\begin{array}{l}
\{\ x = in(T) +\!\!+ \emptyset \quad \wedge \quad y = pre(T)\ \} \\
neg(\emptyset, t) \\
\{\ x = \epsilon \quad \wedge \quad y = \epsilon \quad \wedge \quad t = T\ \}
\end{array}
$$

which is the program we aimed at. Observe that a stack is required to implement the recursion. At any time, the storage requirement for the stack is at most the storage requirement for the part of the tree that has not yet been constructed. Therefore, the algorithm does not require any extra

space for constructing the tree from its traversals. This is also true of the iterative algorithms (cf. [3,6]) in which the stack is explicit.

Acknowledgement

Dong Lin is gratefully acknowledged for "seeing" the recursive algorithm instantaneously in class, forcing me to produce a correctness argument on the spot. David Gries provided helpful comments on the presentation.

References

[1] H.A. Burgdorff, S. Jajodia, F.N. Springsteel, Y. Zalcstein, *Alternative methods for the reconstrcution of trees from their traversals*, BIT vol. 27 (1987) 134–140.

[2] G.-H. Chen, M.S. Yu, L.T. Liu, *Two algorithms for constructing a binary trtee from its traversals*, Information Processing Letters, vol. 28(1988) 297–299.

[3] W. Chen, J.T. Udding, *Program inversion: more than fun!*, Science of computer Programming, vol. 15 (1990), 1–13.

[4] Edsger W. Dijkstra, *Selected Writings on Computing: a Personal Perspective*, Springer-Verlag, 1982.

[5] D. Gries, *The Science of Programming*, Chapter 21: Inverting Programs, Springer-Verlag, 1982.

[6] D. Gries, J.L.A. van de Snepscheut, *Inorder Traversal of a Binary Tree and its Inversion*, in: Formal Development of Programs and Proofs, University of Texas at Austin Year of Programming Series, edited by Edsger W. Dijkstra, Addison-Wesley 1990, p.37–42

[7] D.E. Knuth, *The Art of computer Programming, Vol. 3: Sorting and Searching*, Addison-Wesley, 1973.